Universidad de Valladolid

Facultad de Ciencias

Máster en matemáticas

COMPUTACIÓN PARALELA Y CÁLCULO DISTRIBUIDO

PROYECTO FINAL

Víctor R. Díez Recio

15 de junio de 2025

Índice

1. Introducción

Este trabajo se ha desarrollado con el objetivo de aplicar técnicas de programación paralela utilizando el estándar `MPI` (Message Passing Interface). A través de un enfoque práctico, se ha implementado una versión del clásico juego de *Hundir la flota*, simulando partidas entre diferentes estrategias de juego distribuidas entre procesos paralelos.

El proyecto consiste en desarrollar un simulador donde dos jugadores, representados por procesos distintos, que compiten colocando y disparando a flotas enemigas en tableros independientes. La interacción entre ambos jugadores se realiza exclusivamente a través de comunicaciones `MPI`, asegurando la correcta sincronización de los turnos, el paso de información (como coordenadas de disparo y resultados), y el control del flujo de la partida. Además, se ha incorporado un tercer proceso (coordinador), que es el que gestiona la lógica mas general, centrada en resultados, gestión de estadísticas, visualización, etc.

La implementación del juego se ha realizado en Python con un diseño modular, de manera que se separan las responsabilidades a lo largo de varios scripts para tener un código organizado, limpio y poder probar varias estrategias de juego. De esta forma, se pueden llevar a cabo un gran numero de simulaciones enfrentando distintas estrategias, con el fin de evaluar su comportamiento estadístico.

A lo largo del desarrollo han ido apareciendo diversos problemas relacionados con la sincronización entre procesos y el intercambio de mensajes, que se han ido solucionando poco a poco. Esta memoria recoge no solo los aspectos técnicos del diseño e implementación, sino también las decisiones adoptadas, los problemas encontrados y las soluciones aplicadas durante el proceso.

Para ejecutar el programa, se utiliza el comando `mpiexec` indicando el número de procesos a lanzar. En la configuración final del proyecto, se emplean tres procesos: dos para los jugadores y uno adicional para el coordinador. La línea de ejecución es la siguiente:

```
mpiexec -n 3 python main.py
```

Esto asegura que los dos jugadores se enfrenten en partidas gestionadas por un coordinador central, que recopila estadísticas, visualiza los resultados y mantiene la sincronización entre procesos.

El objetivo final es demostrar cómo una lógica de juego sencilla puede ser abordada desde una perspectiva paralela y distribuida, utilizando herramientas adecuadas, permitiendo así una comprensión más profunda de la programación con `MPI`.

2. Implementación de la paralelización con MPI

La simulación está diseñada para ejecutarse con tres procesos MPI distintos. Dos de ellos representan a los jugadores (con `rank 0` y `rank 1`), y un tercero actúa como proceso coordinador (con `rank 2`).

- **Jugador 0 y Jugador 1**: ejecutan la lógica del juego de forma paralela. Cada uno tiene su propio tablero, flota y estrategia de disparo.
- **Coordinador (rank 2)**: se encarga exclusivamente de lanzar las partidas, distribuir las estrategias, recopilar estadísticas, y generar los resúmenes y visualizaciones globales.

Con esta separación podemos escalar el sistema y mejorar su claridad, reduciendo la carga de trabajo en los procesos de juego.

2.1. Comunicación entre procesos

La comunicación se realiza con la librería de Python `mpi4py`. Cada vez que un jugador realiza un disparo, se envía una tupla con las coordenadas al oponente mediante `comm.send()`. El proceso receptor interpreta estas coordenadas y devuelve el resultado del disparo (`'agua'`, `'tocado'`, `'hundido'` o `'FIN'`). Además, al finalizar una partida, los jugadores envían un diccionario con sus estadísticas al coordinador. El coordinador espera recibir todos los resultados para después analizarlos de forma centralizada.

La sincronización se gestiona a través del número de turno en la variable `turno`, que se incrementa en un proceso y se envía al otro. Esta variable se evalúa con la condición `turno % 2 == rank`, lo que permite que cada jugador sepa cuándo debe disparar y cuándo debe esperar un disparo. De esta forma conseguimos que nunca disparen ambos jugadores a la vez, y que los disparos sean siempre alternos, siguiendo así la dinámica de un juego por turnos.

2.2. Gestión del coordinador (rank 2) y recolección de estadísticas

El proceso coordinador se activa únicamente en `rank == 2`. Actua solo desde el script `main.py` y tiene un rol es crucial para orquestar varias simulaciones sin sobrecargar a los jugadores. Su lógica se resume en:

1. Generar todas las combinaciones posibles de estrategias.
2. Asignar a los jugadores una combinación y esperar el resultado.
3. Recoger estadísticas de la partida y almacenarlas.
4. Una vez finalizadas todas las simulaciones, construir resúmenes, imprimir tablas y generar mapas de calor.

La parte del código donde este proceso gestiona las simulaciones se puede ver en la figura 1.

```python
estrategias = list(ESTRATEGIAS_DISPONIBLES.keys())

if rank == 2:
    resultados = []
    # Proceso maestro: gestiona todas las combinaciones de estrategias y controla los turnos
    for e0 in estrategias:
        for e1 in estrategias:
            print(f"\nSimulando {NUM_SIMULACIONES} partidas entre {e0.upper()} vs {e1.upper()}...\n")
            for _ in range(NUM_SIMULACIONES):
                # En cada partida, se envían las estrategias a rank 0 y 1
                comm.send((e0, e1), dest=0, tag=0)
                comm.send((e0, e1), dest=1, tag=0)

                # Recogemos los resultados desde el rank 0
                resultado = comm.recv(source=0, tag=1)
                resultados.append(resultado)
    # Después de todas las partidas, se manda una señal de parada
    comm.send(None, dest=0, tag=9)
    comm.send(None, dest=1, tag=9)

# Jugadores: ejecutan partidas cuando reciben estrategias del coordinador
else:
    while True:
        status = MPI.Status()
        datos = comm.recv(source=2, tag=MPI.ANY_TAG, status=status)
        if status.Get_tag() == 9:
            break
        e0, e1 = datos
        resultado = jugar_una_partida(e0, e1)
        if rank == 0:
            comm.send(resultado, dest=2, tag=1)
```

Figura 1: código simplificado de `main.py`.

En resumen, se crea una lista de las estrategias `ESTRATEGIAS_DISPONIBLES` que se van a seguir y se enfrentan dos jugadores con las estrategias definidas en dos a dos. Esta centralización permite que los procesos de juego se mantengan ligeros y únicamente centrados en la dinámica del juego, llamándolos cada vez que comience una nueva partida.

3. Diseño general del software

Como hemos mencionado anteriormente, el proyecto se ha organizado de forma modular para facilitar su comprensión, mantenimiento y posible extensión. Esto ha permitido añadir varias estrategias de juego de manera mucho mas sencilla y limpia, cuidar y revisar ciertos aspectos o lógicas de juego de manera mucho más cómoda y poder corregir bugs o errores mucho más fácil. Cada script tiene una responsabilidad clara:

- `main.py`: es el punto de entrada del programa. Se encarga de coordinar las simulaciones múltiples entre combinaciones de estrategias, y centraliza los resultados.
- `partida.py`: es el encargado de gestionar la lógica de una partida individual entre dos jugadores usando MPI. Aquí se controla el flujo de turnos, el intercambio de disparos y los mensajes entre procesos.
- `jugador.py`: define el comportamiento de un jugador, incluyendo la estrategia, como maneja su tablero y la lógica de respuesta a los disparos recibidos.
- `tablero.py` y `flota.py`: se encargan de la generación del tablero, de la colocación de barcos y de como se procesan de impactos en la flota.
- `constantes.py`: aquí se definen las constantes globales que se utilizan en el proyecto (símbolos de tablero, tamaño, número de simulaciones, etc.). De esta forma, es mucho más sencillo hacer pruebas e ir cambiando parámetros.
- `estrategias/`: esta carpeta contiene todas las estrategias de disparo, que figuran como clases, ya que es una forma muy sencilla y cómoda de crearlas. Cada estrategia implementa una lógica distinta de búsqueda y ataque, aunque todas heredan de una base, en `estrategias/base.py`. En total son 3 distintas, que se explicarán más adelante en la sección 5.

Con esta estructura se puede mantener una separación clara y limpia entre la lógica del juego y la lógica de paralelización con MPI.

3.1. Lógica general de ejecución

El programa comienza ejecutando `main.py`, donde el tercer proceso (Rank 2) lanza las simulaciones estipuladas para cada una de todas las combinaciones posibles de estrategias jugador vs jugador con la función `jugar_una_partida` del script `partida.py`. Por cada combinación, actuan estos otros dos procesos restantes:

- Rank 0: representa al Jugador 0.
- Rank 1: representa al Jugador 1.

Nota. Normalmente el proceso 0 suele ser el coordinador, pero en este caso, la decisión de este tercer nodo coordinador se tomo al final, por lo que para no cambiar todo el código y dar lugar a errores, finalmente el rol lo tomo el tercer proceso (Rank 2).

Cuando se llama a la función `jugar_una_partida`, cada jugador genera su tablero, coloca su flota de barcos y comienza una partida con la estrategia configurada en la que se alternan

los turnos. En cada turno, un jugador envía sus coordenadas de disparo al otro proceso, que responde con el resultado del impacto. La partida continúa hasta que uno de los jugadores pierde toda su flota. El código principal dentro de `jugar_una_partida`, en esencia, es el que se puede ver en la figura 2.

```python
# === Bucle principal del juego ===
while not juego_terminado:

    # === TURNO DE REALIZAR DISPARO ===
    if turno % 2 == rank:
        # Dispara el jugador que le toca y le manda las coordenadas al otro jugador
        x, y = jugador.siguiente_disparo()
        comm.send((x, y), dest=1 - rank, tag=0)

        # Recibimos la información del jugador que encaja el disparo
        respuesta = comm.recv(source=1 - rank, tag=1)

        # Terminamos el juego si el otro jugador nos indica que hemos
        # hundido su último barco
        if respuesta == "FIN":
            juego_terminado = True
            ganador = rank

    # === TURNO DE RECIBIR DISPARO ===
    else:
        # En el caso de recibir disparo, el jugador disparado recoge las coordenadas que
        # le manda el jugador atacante y responde si le ha sido agua, tocado, hundido
        # o FIN (si ha perdido toda su flota)
        x, y = comm.recv(source=1 - rank, tag=0)
        resultado = jugador.recibir_disparo(x, y)
        comm.send(resultado, dest=1 - rank, tag=1)

        if resultado == "FIN":
            juego_terminado = True

    # Actualizamos el turno: rank 0 incrementa el turno y lo envía a rank 1,
    # este lo recibe y lo actualiza
    if rank == 0:
        turno += 1
        comm.send(turno, dest=1, tag=99)
    else:
        turno = comm.recv(source=0, tag=99)
```

Figura 2: bucle principal resumido del juego en cada simulación, dentro de `jugar_una_partida`.

La comunicación entre los procesos se basa en el modelo maestro-esclavo y punto a punto de `MPI`. Por cada turno, un proceso entrará en el `if` de realizar disparo y el otro en el de recibir disparo. Llamemos a estos dos bloques *Acción atacante* y *Acción defensor* respectivamente. Una vez se realicen todas las acciones, el primer proceso avanza un turno y se lo envía al segundo proceso, para que sea solo uno el que lleve la cuenta y así no dar lugar a errores. Por cada turno, el flujo es el siguiente:

1. El proceso que tiene el turno de disparar (según el turno global) entra en el bloque *Acción atacante*, genera las coordenadas `(x, y)` a donde quiere disparar con el método `siguiente_disparo` y se las envía al proceso oponente con `comm.send(..., dest=..., tag=0)`.

2. El receptor, que habrá entrado en el bloque *Acción defensor*, procesa el disparo con el método `recibir_disparo` y responde con `agua`, `tocado`, `hundido` o `FIN`, dependiendo de si se golpea a un barco y si quedan barcos sin golpear.

3. Si se ha activado el guardado de eventos, ambos procesos pueden enviar al proceso coordinador (Rank 2) información sobre la jugada. Esto se hace para procesar las jugadas relevantes y no almacenar demasiada información innecesaria en simulaciones grandes. Esta parte no aparece en el bucle resumido de la figura 2 por simplicidad.

4. Una vez realizado este proceso, el primer proceso avanza un turno y le envía el número al segundo proceso, para que el bucle vuelva a empezar si aún quedan barcos en ambos tableros.

5. Al finalizar la partida, se recopilan estadísticas y se transmite todo al coordinador para mostrar por pantalla los resultados.

Nota. El método `siguiente_disparo` es diferente dependiendo de la estrategia que esté siguiendo cada jugador. Los diferentes métodos para generar el disparo se explican mas detalladamente en la sección 5.

Esto garantiza que la lógica de cada jugador sea independiente, pero esté sincronizada mediante mensajes MPI en cada turno.

Algunas decisiones clave que se tomaron durante el desarrollo son:

- Usar un proceso adicional (Rank 2) como coordinador, para descargar de trabajo al proceso Rank 0 y sobretodo evitar evitar conflictos a la hora de recolectar y procesar los eventos.

- Implementar las estrategias como clases independientes, de manera que se pueden configurar en cada jugador al inicio de la partida, para permitir fácilmente pruebas comparativas.

- Diseñar un sistema de eventos de tablero que recoja sólo los movimientos relevantes (modo caza, cambio de estado, hundimiento), y almacenarlos para ser impresos en orden al final de la partida. Estos se explicarán mas adelante en la sección 5.

- Controlar los turnos mediante una única variable global compartida entre los procesos sincronizados mediante el valor `rank`. Solo la procesa un solo proceso para evitar conflictos a la hora de entrar en el bloque *Acción atacante* o *Acción defensor*.

Estas decisiones hacen mucho más fácil la paralelización efectiva y la posibilidad de extender el sistema en un futuro.

3.2. Problemas encontrados y soluciones

Durante el desarrollo, surgieron varios problemas:

- **Duplicación de turnos:** en algunos casos, los jugadores disparaban varias veces en el mismo turno. Esto se solucionó con un control más estricto del valor de `turno` y asegurando que sólo el primer proceso lleve la cuenta y se la envíe al final de cada turno al segundo proceso.
- **Disparos repetidos:** algunos disparos se repetían debido a que las estrategias no registraban correctamente los disparos anteriores. Se solucionó añadiendo estructuras auxiliares como sets de disparos realizados, listas de barcos hundidos o eventos significativos para reforzar el control sobre la validez de acciones, mejorar el comportamiento de las estrategias, y hacer que la traza del juego sea más fácil de seguir.
- **Desincronización al terminar la partida:** al principio, sólo el jugador que disparaba conocía el fin del juego. Se mejoró para que ambos procesos compartan el estado de fin y puedan salir del bucle a la vez.
- **Congelación o espera infinita:** cuando un proceso esperaba un mensaje que nunca llegaba por mal control de flujos. Se solucionó afinando las condiciones de fin y gestionando correctamente los mensajes de cierre.

Tras solucionar los problemas se consiguió afinar y estabilizar el programa, y fortalecer la comprensión del modelo de paso de mensajes de MPI.

4. Descripción de los módulos del software

Pasemos ahora a describir más detalladamente el código de cada uno de los cripts del programa.

4.1. main.py: punto de entrada y orquestación de simulaciones

Como ya hemos visto previamente, este módulo actúa como punto de entrada del programa. Se encarga de inicializar el entorno MPI y coordinar el lanzamiento de múltiples partidas entre diferentes estrategias.

En el caso del proceso con `rank = 2` (coordinador), se generan todas las combinaciones posibles de estrategias y se lanzan simulaciones en paralelo. Después se recogen los resultados y se presenta un resumen detallado, incluyendo tablas de rendimiento y matrices de calor.

Los procesos con `rank = 0` y `rank = 1` funcionan como jugadores. Esperan instrucciones del coordinador para comenzar partidas y ejecutan la lógica de juego según la estrategia asignada.

4.2. partida.py: lógica principal del juego y la comunicación MPI

Este módulo contiene la función `jugar_una_partida()`, que representa el núcleo del juego para un par de jugadores.

Su principal responsabilidad es orquestar el turno a turno, asegurando que se respete correctamente la alternancia entre jugadores. También gestiona la comunicación entre procesos: uno envía el disparo, el otro lo procesa y responde con el resultado.

Incluye además la recolección de estadísticas, el registro de eventos relevantes, la impresión por pantalla del resultado de disparos y el aspecto del tablero por turno y la lógica para finalizar la partida cuando uno de los jugadores ha perdido todos sus barcos.

4.3. jugador.py: representación y comportamiento de cada jugador

Define la clase `Jugador`, que encapsula toda la información relativa a un jugador individual: su tablero, su flota, su estrategia de disparo, y las funciones necesarias para procesar disparos entrantes y generar nuevos disparos.

Destacan los métodos `siguiente_disparo()`, `registrar_resultado_disparo()` y `recibir_disparo()` que son los que mantienen el estado del jugador sincronizado con la evolución de la partida. Estos pueden variar dependiendo el tipo de estrategia.

4.4. tablero.py y flota.py: lógica de juego y utilidades

tablero.py contiene funciones auxiliares para imprimir el tablero y marcar impactos de forma visual. **flota.py** se encarga de la generación aleatoria de la flota de barcos. Se asegura de que los barcos no se solapen ni se coloquen demasiado cerca unos de otros, respetando la distancia mínima establecida.

Ambos módulos son claves para mantener la lógica de juego clara y modular.

4.5. constantes.py: configuración del juego

Este archivo contiene todos los parámetros que se pueden configurar. Son los siguientes:

- `BOARD_SIZE`: tamaño del tablero cuadrado (número de filas y columnas). Por defecto, tenemos un tablero 20×20.
- `NUM_SIMULACIONES`: número de simulaciones que se ejecutan por combinación de estrategias. Permite definir la carga del experimento. Por defecto, `1`, que correspondería a realizar un solo juego por combinación de estrategias.
- `ESTRATEGIAS_DISPONIBLES`: diccionario con las estrategias disponibles que se van a procesar. Permite añadir fácilmente nuevas estrategias sin modificar el flujo principal.
- `MOSTRAR_DISPAROS`: controla el nivel de detalle para imprimir disparos durante la ejecución. Puede tomar los valores: `"Todos"`, `"Solo aciertos"`, o `"Ninguno"`.
- `MOSTRAR_TABLERO`: valor booleano que activa o desactiva la impresión del tablero durante el juego. Es útil para depurar.
- `TAMANOS_BARCOS`: lista de enteros que define los tamaños de los barcos que forman la flota. Por defecto tenemos un barco de tamaño 5, uno de 4, dos de 3 y uno de 2: `[5, 4, 3, 3, 2]`.
- `SIMBOLO_VACIO`, `SIMBOLO_BARCO`, `SIMBOLO_AGUA`, `SIMBOLO_TOCADO`: caracteres que representan visualmente el estado de cada celda del tablero. Se pueden personalizar para adaptar la salida.

Esto permite ajustar rápidamente el comportamiento del juego sin modificar el código, que hace más fácil el poder experimentar.

4.6. Estrategias: definición y funcionamiento de estrategias

Las estrategias están organizadas en archivos separados dentro de la carpeta `estrategias/`. Siguen una interfaz común, siendo clases heredadas de una `Estrategia`, en `base.py`. Cada estrategia implementa:

- `siguiente_disparo()`: devuelve las coordenadas del próximo disparo.
- `registrar_resultado()`: actualiza su lógica interna con el resultado obtenido.

El sistema está diseñado para facilitar la incorporación de nuevas estrategias en el futuro, simplemente añadiendo nuevas clases que hereden de la clase base `Estrategia`. Se explican con más detalle en la sección 5.

5. Estrategias implementadas

El proyecto incluye un total de tres estrategias, que como ya hemos mencionado, consisten en clases que heredan de la clase base `Estrategia`, ubicada en `estrategias/base.py`. Cada estrategia implementa una forma distinta de decidir la próxima coordenada donde disparar, aunque todas verifican que la coordenada no haya sido disparada antes ni esté fuera de los límites del tablero. Pasemos a definirlas.

5.1. Estrategia `Aleatoria`

Esta estrategia tiene es la más simple posible: en cada turno, selecciona al azar una casilla del tablero que aún no haya sido atacada. No utiliza información anterior ni ajusta su comportamiento según los resultados obtenidos.

Aunque es útil como punto de referencia para comparar contra estrategias más sofisticadas, su rendimiento es claramente inferior, especialmente cuando se enfrenta a estrategias con comportamiento adaptativo.

5.2. Estrategia `Optimizada`

Esta estrategia implementa dos modos de operación bien diferenciados: **modo exploración** y **modo caza**, con transiciones dinámicas entre ambos según el resultado del disparo anterior.

- **Modo exploración:** el jugador dispara aleatoriamente en el tablero (sin repetir posiciones), con el objetivo de encontrar cualquier parte de un barco enemigo. Se basa en la suposición de que es necesario primero impactar al menos una vez para poder optimizar la posterior secuencia de disparos.
- **Transición a caza:** si un disparo resulta en `tocado`, se activa el modo caza.
- **Modo caza:** el jugador se centra en las casillas vecinas del impacto inicial. En un primer momento se exploran las posiciones ortogonales adyacentes (arriba, abajo, izquierda, derecha) al primer `tocado`. Cuando se produce un segundo impacto, entonces ya se conoce la dirección del barco (horizontal o vertical), por lo se continúa disparando en esa dirección hasta hundirlo completamente. Una vez que el barco se considera `hundido`, se limpian las listas de posiciones `tocado` y `candidatos`, y se retorna al modo exploración.

Este diseño emula el comportamiento racional de un jugador humano. Gracias a estas mejoras, `Optimizada` incrementa la velocidad para hundir los barcos encontrados, mejorando notablemente la eficiencia respecto a estrategias puramente aleatorias. De esta manera que se acorta significativamente el número total de turnos para finalizar una partida.

5.3. Estrategia `Optimizada2`

`Optimizada2` es una mejora directa sobre la estrategia anterior. Conserva los dos modos fundamentales (*exploración* y *caza*), pero introduce dos optimizaciones clave:

- **Exploración con patrón de ajedrez:** durante el modo exploración, en lugar de seleccionar posiciones aleatorias arbitrarias, se impone una condición de paridad sobre las coordenadas (patrón de ajedrez), seleccionando solo las casillas donde $(x+y) \mod 2 = 0$.

Esto permite cubrir el tablero de forma más eficiente, ya que ningún barco puede ocupar una sola casilla y todos tienen al menos longitud 2, por lo que pueden ser interceptados con esta técnica.

- **Zonas de exclusión alrededor de barcos hundidos:** para evitar disparar cerca de zonas que ya han sido exploradas con éxito, se descartan todas las celdas adyacentes (incluyendo diagonales) a cualquier casilla que haya formado parte de un barco ya `hundido`. Esto se basa en las reglas del juego que prohíben colocar barcos adyacentes unos a otros, creando así una "burbuja de seguridad" alrededor de cada barco destruido.
- **Caza como en la versión anterior:** al entrar en modo caza, se aplican las mismas reglas de generación y seguimiento de candidatos, extendiendo en la dirección deducida hasta que el barco sea hundido por completo.

Gracias a estas mejoras, `Optimizada2` incrementa la velocidad para encontrar barcos. Emulando partidas se muestra que esta estrategia supera claramente a sus predecesoras en número de turnos y porcentaje de victorias frente a oponentes menos sofisticados.

6. Resultados de las simulaciones

Para evaluar la efectividad de las distintas estrategias, se desarrolló una infraestructura que permite simular automáticamente miles de partidas entre combinaciones posibles de estrategias. Las simulaciones se ejecutaron utilizando MPI con los tres procesos descritos previamente: dos jugadores y un coordinador. El coordinador recopila los resultados y calcula estadísticas relevantes de cada enfrentamiento.

Cada combinación estrategia vs estrategia fue ejecutada **50,000 veces**, lo cual proporciona resultados estadísticamente significativos. Esto da lugar a 9 combinaciones distintas de enfrentamientos (jugador 0 vs jugador 1), con 50,000 partidas simuladas por combinación, resultando en un total de 450,000 partidas.

6.1. Análisis de resultados

En la tabla 1, se muestra un resumen detallado de los resultados. Se incluyen métricas como precisión de disparos, número de turnos promedio y porcentaje de victorias por jugador.

Tabla 1: Resumen de estadísticas por enfrentamiento entre estrategias

J0	J1	%J0	%J1	Turnos	Prec. J0	Prec. J1	Disp. J0	Disp. J1
alea	alea	51.2	48.8	735.5	4.2 %	4.2 %	368.0	367.5
alea	opt	0.8	99.2	423.5	4.3 %	7.6 %	211.8	211.7
alea	opt2	0.0	100.0	309.3	4.2 %	10.3 %	154.6	154.6
opt	alea	99.1	0.9	424.1	7.5 %	4.2 %	212.5	211.5
opt	opt	49.8	50.2	347.3	8.2 %	8.2 %	173.9	173.4
opt	opt2	24.0	76.0	291.0	8.9 %	10.4 %	145.6	145.4
opt2	alea	100.0	0.0	307.7	10.4 %	4.2 %	154.3	153.3
opt2	opt	76.7	23.3	290.1	10.4 %	8.9 %	145.4	144.7
opt2	opt2	50.7	49.3	272.5	10.5 %	10.5 %	136.5	136.0

Para facilitar la interpretación visual de las tasas de victoria en cada enfrentamiento, se ha generado un mapa de calor (figura 3). Cada celda representa el porcentaje de partidas ganadas por el Jugador 1 en la combinación dada, con colores que van del azul (0 %) al rojo (100 %).

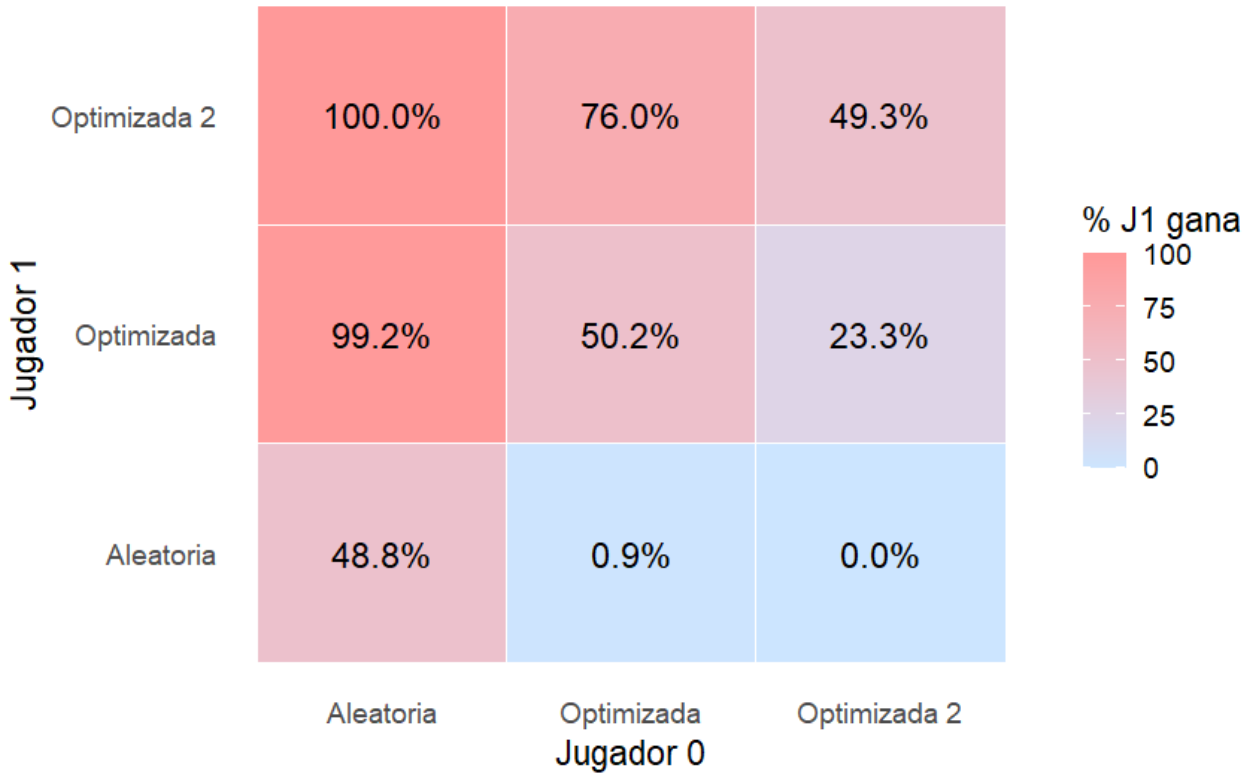

Figura 3: matriz de calor del porcentaje de victorias del Jugador 1 por enfrentamiento

Del análisis podemos sacar las siguientes conclusiones:

- La estrategia `aleatoria` es ampliamente superada por las demás, como cabría esperar.
- La `optimizada2`, con ajustes más agresivos, ofrece los mejores resultados generales, ganando en la mayoría de partidas contra la `optimizada`.
- En enfrentamientos entre estrategias similares (como `opt` vs `opt`), los resultados tienden a estar equilibrados, reflejando que no existe una ventaja táctica clara cuando ambas usan los mismos mecanismos.
- Parece que la precisión de disparo varía significativamente entre las distintas estrategias implementadas, reflejando el nivel de inteligencia de cada una. La estrategia *aleatoria* apenas alcanza un porcentaje de acierto medio del 4.2 %, mientras que la estrategia *optimizada* mejora este valor situándose entre el 7.5 % y el 8.9 %. Por otro lado, la *optimizada2* logra la mayor eficiencia, con una tasa de acierto media que ronda el 10.3–10.5 %.

7. Conclusiones

El desarrollo de este proyecto ha concluido con la exitosa implementación práctica de técnicas de computación paralela, específicamente mediante el uso de MPI (Message Passing Interface). Más allá de los aspectos de la dificultad de implementar la lógica del juego de hundir la flota, el objetivo principal era construir una arquitectura distribuida en la que dos procesos se comportaran como jugadores autónomos, intercambiando información bajo un esquema de paso de mensajes síncronizado y controlado.

A lo largo del trabajo, se ha profundizado en aspectos clave de la programación paralela: la sincronización de procesos, la comunicación punto a punto, el diseño modular y la optimización del flujo de datos para evitar cuellos de botella y duplicidades.

7.1. Fortalezas y debilidades del enfoque MPI usado

Entre los aspectos que mejor han funcionado en el diseño del proyecto, destaca la separación clara entre los distintos roles de los procesos, especialmente la inclusión de un tercer proceso que actúa como coordinador (`rank 2`), encargado exclusivamente de gestionar las estadísticas y mostrar los resultados. Esto ha permitido simplificar bastante la lógica de los jugadores y mantener el código más limpio. Además, el uso de turnos estrictos controlados por el número de iteración ha sido clave para que los jugadores se alternen correctamente sin pisarse, evitando así errores difíciles de depurar. También ha sido muy útil haber dividido el proyecto en varios módulos independientes, lo que ha hecho más sencilla su evolución y análisis.

Por otro lado, el enfoque también presenta algunas limitaciones. El tipo de comunicación entre procesos usado, aunque más fácil de implementar y seguir, implica que cada proceso debe esperar a que el otro termine su parte, lo que puede suponer un freno si uno de ellos se retrasa. Además, a medida que se han ido añadiendo nuevas funcionalidades, el número de mensajes intercambiados ha ido creciendo, haciendo el código más complejo.

7.2. Posibles mejoras o líneas futuras

A partir de esta base, hay varias direcciones en las que el proyecto podría seguir creciendo. Una de ellas sería rediseñar el esquema de comunicación para hacerlo más flexible, permitiendo que los jugadores trabajen en paralelo y solo se coordinen cuando realmente sea necesario. También sería interesante mejorar el rol del coordinador para que recoja más datos y genere informes automáticos sobre las partidas. En el plano de las estrategias, se podría avanzar hacia enfoques más inteligentes, incluso aplicando técnicas de aprendizaje automático para que el sistema se convierta en un entorno de pruebas para algoritmos de decisión, aunque no es el objetivo principal del trabajo. Por último, una ampliación podría consistir en permitir partidas con más de dos jugadores y organizar torneos completos.

En resumen, este trabajo demuestra que es perfectamente posible aplicar MPI para gestionar de forma ordenada la interacción entre distintos jugadores, incluso en un juego como este. La clave ha estado en diseñar bien la estructura desde el principio, lo que ha hecho que el sistema sea sólido, entendible y fácil de extender.